Kam1n0: MapReduce-based Assembly Clone Search for Reverse Engineering

Steven H. H. Ding† Benjamin C. M. Fung∗ Philippe Charland†

†School of Information Studies, McGill University, Montreal, QC, Canada
∗Mission Critical Cyber Security Section, Defence R&D Canada - Valcartier, Quebec, QC, Canada
steven.h.ding@mail.mcgill.ca ben.fung@mcgill.ca philippe.charland@drdc-rddc.gc.ca

ABSTRACT

Assembly code analysis is one of the critical processes for detecting and proving software plagiarism and software patent infringements when the source code is unavailable. It is also a common practice to discover exploits and vulnerabilities in existing software. However, it is a manually intensive and time-consuming process even for experienced reverse engineers. An effective and efficient assembly code clone search engine can greatly reduce the effort of this process, since it can identify the cloned parts that have been previously analyzed. The assembly code clone search problem belongs to the field of software engineering. However, it strongly depends on practical nearest neighbor search techniques in data mining and databases. By closely collaborating with reverse engineers and Defence Research and Development Canada (DRDC), we study the concerns and challenges that make existing assembly code clone approaches not practically applicable from the perspective of data mining. We propose a new variant of LSHP scheme and incorporate it with graph matching to address these challenges. We implement an integrated assembly clone search engine called Kam1n0. It is the first clone search engine that can efficiently identify the given query assembly function’s subgraph clones from a large assembly code repository. Kam1n0 is built upon the Apache Spark computation framework and Cassandra-like key-value distributed storage. A deployed demo system is publicly available.1 Extensive experimental results suggest that Kam1n0 is accurate, efficient, and scalable for handling large volume of assembly code.

1. INTRODUCTION

Code reuse is a common but uncontrolled issue in software engineering [15]. Mockus [25] found that more than 50% of files were reused in more than one open source project. Sojer’s survey [20] indicates that more than 50% of the developers modify the components before reusing them. This massively uncontrolled reuse of source code does not only introduce legal issues such as GNU General Public License (GPL) violation [36, 17]. It also implies security concerns, as the source code and the vulnerabilities are uncontrollably shared between projects [4].

Identifying all these infringements and vulnerabilities requires intensive effort from reverse engineers. However, the learning curve to master reverse engineering is much steeper than for programming [4]. Reverse engineering is a time consuming process which involves inspecting the execution flow of the program in assembly code and determining the functionalities of the components. Given the fact that code reuse is prevalent in software development, there is a pressing need to develop an efficient and effective assembly clone search engine for reverse engineers. Previous clone search approaches only focus on the search accuracy. However, designing a practically useful clone search engine is a non-trivial task which involves multiple factors to be considered. By closely collaborating with reverse engineers and Defence Research and Development Canada (DRDC), we outline the deployment challenges and requirements as follows:

Interpretability and usability: An assembly function can be represented as a control flow graph consisting of connected basic blocks. Given an assembly function as query, all of the previous assembly code clone search approaches [7, 6, 18, 26] only provide the top-listed candidate assembly functions. They are useful when there exists a function in the repository that shares a high degree of similarity with the query. However, due to the unpredictable effects of different compilers, compiler optimization, and obfuscation techniques, given an unknown function, it is less probable to have a very similar function in the repository. Returning a list of clones with a low degree of similarity values is not useful. As per our discussions with DRDC, a practical search engine should be able to decompose the given query assembly function into different basic blocks which can help reverse engineers better understand the function’s composition. We define a subgraph clone as one of its subgraphs that can be found in the other function. Refer to the example in Figure 1. The previous clone search approaches cannot address this challenge.

Keywords

Assembly clone search; Information retrieval; Mining software repositories

---

1Kam1n0 online demo (no installation required). Both the user name and password are “sigkdd2016”. Use Chrome for best experience. http://dmas.lab.mcgill.ca/projects/kam1n0.htm

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

KDD '16 August 13–17, 2016, San Francisco, CA, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4232-2/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2939672.2939719

Figure 1: An example of the clone search problem. Basic blocks with a white background form a subgraph clone between two functions. Three types of code clones are considered in this paper. Type I: literally identical; Type II: syntactically equivalent; and Type III: minor modifications.

Efficiency and Scalability: An efficient engine can help reverse engineers retrieve clone search results on-the-fly when they are conducting an analysis. Instant feedback tells the reverse engineer the composition of a given function that is under investigation. Scalability is a critical factor as the number of assembly functions in the repository needs to scale up to millions. The degradation of search performance against the repository size has to be considered. Previous approaches in [6, 7], which trade efficiency and scalability for better accuracy, have a high latency for queries and are thus not practically applicable.

Incremental updates: The clone search engine should support incremental updates to the repository, without re-indexing existing assembly functions. [7] requires median statistics to index each vector and [26] requires data-dependent settings for its index, so they do not satisfy this requirement.

Clone search quality: Practically, clones among assembly functions are one-to-many mappings, i.e., a function has multiple cloned functions with different degrees of similarity. However, all previous approaches [6, 7, 18, 26] assume that clones are one-to-one mappings in the experiment. This is due to the difficulty of acquiring such a one-to-one labeled dataset. Moreover, they use different evaluation metrics. Therefore, it is difficult to have a direct comparison among them with respect to the search quality. We need to develop a one-to-many labeled dataset and an unified evaluation framework to quantify the clone search quality.

To address the above requirements and challenges, we propose a new variant of LSH scheme and incorporate it with a graph matching technique. We also develop and deploy a new assembly clone search engine called Kam1n0. Our major contributions can be summarized as follows:

- **Solved a challenging problem for the reverse engineering community:** Kam1n0 is the first assembly code clone search engine that supports subgraph clone search. Refer to the example in Figure 1. It promotes the interpretability and usability by providing subgraph clones as results, which helps reverse engineers analyzing new and unknown assembly functions. Kam1n0 won the second prize at the 2015 Hex-Rays plugin Contest and its code is publicly accessible on GitHub.

- **Efficient inexact assembly code search:** The assembly code vector space is highly skewed. Small blocks tend to be similar to each other and large blocks tend to be sparsely distributed in the space. Original hyperplane hashing with banding technique equally partitions the space and does not handle the unevenly distributed data well. We propose a new adaptive locality sensitive hashing (ALSH) scheme to approximate the cosine similarity. To our best knowledge, ALSH is the first incremental locality sensitive hashing scheme that solves this issue specifically for cosine space with theoretical guarantee. It retrieves fewer points for dense areas and more points for sparse ones in the cosine space. It is therefore efficient in searching nearest neighbors.

- **Scalable sub-linear subgraph search:** We propose a MapReduce subgraph search algorithm based on the Apache Spark computational framework without an additional index. Different to the existing subgraph isomorphism search problem in data mining, we need to retrieve subgraphs that are both isomorphic to the query and the repository functions as graphs. Thus, existing algorithms are not directly applicable. Algorithmically, our approach is bounded by polynomial complexity. However, our experiment suggests that it is sub-linear in practice.

- **Accurate and robust function clone search:** Kam1n0 is the first approach that integrates both the inexact assembly code and subgraph search. Previous solutions do not consider both of them together. Our extensive experiments suggest that it boosts the clone search quality and yields stable results across different datasets and metrics.

- **Develop a labeled dataset and benchmark state-of-the-art assembly code clone solutions.** We carefully construct a new labeled one-to-many assembly code clone dataset that is available to the research community by linking the source code and assembly function level clones. We benchmark and report the performance of twelve existing state-of-the-art solutions with Kam1n0 on the dataset using several metrics. We also setup a mini-cluster to evaluate the scalability of Kam1n0.

The remainder of this paper is organized as follows. Section 2 situates our study within the literature of three different research problems. Section 3 formally defines the studied problem. Section 4 provides an overview of our solution and system design. Section 5 presents the preprocessing steps and the chosen vector space. Section 6 introduces our proposed locality sensitive hashing scheme. Section 7 presents our graph search algorithm. Section 8 presents our benchmark experiments. Section 9 provides the conclusion.

---

2[https://hex-rays.com/contests/2015/#kam1n0](https://hex-rays.com/contests/2015/#kam1n0)
3[https://github.com/MrGill-DMaS/Kam1n0-Plugin-IDA-Pro](https://github.com/MrGill-DMaS/Kam1n0-Plugin-IDA-Pro)
2. RELATED WORK

Locality sensitive hashing. Locality sensitive hashing (LSH) has been studied for decades to solve the \( \epsilon \)-approximated Nearest Neighbor (\( \epsilon \text{NN} \)) problem, since exact nearest neighbor is not scalable to high dimensional data. One of the prevalent problems of LSH is the uneven data distribution issue, as LSH equally partitions the data space. To mitigate this issue, several approaches have been proposed including the LSH-Forest [2], LSB-Forest [32], C2LSH [10] and SK-LSH [22]. It has been shown that the cosine vector space is robust to different compiler settings [18] in assembly code clone search. However, LSH-Forest, C2LSH and SK-LSH are designed for the \( p \)-stable distribution, which does not fit the cosine space. LSH-Forest dynamically and unequally partition the data space. As pointed out by [33], it requires the hash family to possess the \((\epsilon, f(\epsilon))\) property. However, to our best knowledge, such a family in cosine space is still unknown. There are other learning-based approaches [11], which do not meet our incremental requirement. Wang et al. [35] provide a more comprehensive survey on LSH. To satisfy our requirements, we propose the ALSH scheme specifically for the cosine space. Different to the LSH-Forest, ALSH takes more than one bit when going down the tree structure and does not require the \((\epsilon, f(\epsilon))\) property for the LSH family to have theoretical guarantee. Unlike LSB forest [2], we dynamically construct the buckets to adapt to different data distributions.

Subgraph isomorphism. Ullmann [34] proposed the first practical subgraph isomorphism algorithm for small graphs. Several approaches were proposed afterwards for large scale graph data, such as TurboISO [13] and STwig [31]. It has been shown that they can solve the subgraph isomorphism problem in a reasonable time. However, they do not completely meet our problem settings. The subgraph isomorphism problem needs to retrieve subgraphs that are isomorphic to the graphs in the repository and identical to the query. However, we need to retrieve subgraphs that are isomorphic to the graphs in the repository and isomorphic to the graph of the query, which significantly increases the complexity. More details will be discussed in Section 7. Such a difference requires us to propose a specialized search algorithm. Lee et al. [21] provide a comprehensive survey and performance benchmark on subgraph isomorphism.

Assembly code clone search. The studies on the assembly code clone search problem are recent. Only a few approaches exist [6, 7, 18, 26]. They all rely on the inexact text search techniques of data mining. BinClone [7] models assembly code into an Euclidean space based on frequency values of selected features. It is inefficient and not scalable due to the exponential 2-combination of features which approximates the 2-norm distance. LSH-S in [26] models assembly code into a cosine space based on token frequency and approximate the distance by hyperplane hashing and banding scheme. It equally partitions the space and suffers from the uneven data distribution problem. Graphlet [18] models assembly code into a cosine space based on extracted signatures from assembly code. However, it cannot detect any subgraph clones that is smaller than the graphlet size. Tracellet [6] models assembly code according to string editing distance. It compares function one by one, which is not scalable. Kam1n0 is fundamentally different to the previous approaches. It is an integration of inexact assembly code

3. PROBLEM STATEMENT

Reverse engineering starts from a binary file. After being unpacked and disassembled, it becomes a list of assembly functions. In this paper, function represents an assembly function; block represents a basic block; source function represents the actual function written in source code, such as C++; repository function stands for the assembly function that is indexed inside the repository; target function denotes the assembly function that is given as a query; and correspondingly, repository blocks and target blocks refer to their respective basic blocks. Each function \( f \) is represented as a control flow graph denoted by \((B, E)\), where \( B \) indicates its basic blocks and \( E \) indicates the edges that connect the blocks. Let \( B(RP) \) be the complete set of basic blocks in the repository and \( F(RP) \) be the complete set of functions in the repository. Given an assembly function, our goal is to search all its subgraph clones inside the repository RP.

Formally, define the search problem as follows:

**Definition 1.** (Assembly function subgraph clone search) Given a target function \( f_t \) and its control flow graph \((B_t, E_t)\), the search problem is to retrieve all the repository functions \( f_s \in RP \), which share at least one subgraph clone with \( f_t \). The shared list of subgraph clones between \( f_s \) and \( f_t \) is denoted by \( sg_t[a : a] \), where \( sg_t[a] \) represents one of them. A subgraph clone is a set of basic block clone pairs \( sg_t[a] = \{(b_t, b_s) \ldots\} \) between \( f_s \) and \( f_t \), where \( b_t \in B_t \), \( b_s \in B_s \), and \((b_t, b_s)\) is a type I, type II, or type III clone (see Figure 1). Formally, given \( f_t \), the problem is to retrieve all \( \{f_s \in RP | |sg_t[a]| > 0\} \). □

4. OVERALL ARCHITECTURE

The Kam1n0 engine is designed for general key-value storage and the Apache Spark\(^4\) computational framework. Its solution stack, as shown in Figure 2, consists of three layers. The data storage layer is concerned with how the data is stored and indexed. The distributed/local execution layer manages and executes the jobs submitted by the Kam1n0 engine and the graph search and the subgraph search. It enables clone subgraph search of any size.

engine. The Kam1n0 engine splits a search query into multiple jobs and coordinates their execution flow. It also provides the RESTful APIs. We have implemented a web-based user interface and an Hex-Rays IDA Pro plugin\(^5\) as clients. IDA Pro is a popular interactive disassembler that is used by reverse engineers.

Figure 3 depicts the data flow of the clone search process. It consists of the following steps. **Preprocessing:** After parsing the input (either a binary file or assembly functions) into control flow graphs, this step normalizes assembly code into a general form, which will be elaborated in Section 5. **Find basic blocks clone pairs:** Given a list of assembly basic blocks from the previous step, it finds all the clone pairs of blocks using ALSH. **Search the subgraph clones:** Given the list of clone block pairs, the MapReduce module merges and constructs the subgraph clones. Note that this clone search process does not require any source code.

### 5. PREPROCESSING AND VECTOR SPACE

We choose the cosine vector space to characterize the semantic similarity of assembly code. It has been shown that the cosine vector space is robust to different compiler settings [18]. It can mitigate the linear transformation of assembly code. For example, to optimize the program for speed, the compiler may unroll and flatten a loop structure in assembly code by repeating the code inside the loop multiple times. In this case, the cosine similarity between the unrolled and original loop is still high, due to the fact that the cosine distance only considers the included angle between two vectors. The features selected in Kam1n0 include mnemonics, combinations of mnemonics and operands, as well as mnemonics n-gram, which are typically used in assembly code analysis [7, 26]. The equivalent assembly code fragments can be represented in different forms. To mitigate this issue, we normalize the operands in assembly code during the preprocessing. We extend the normalization tree used in BinClone [7] with more types. There are three normalization levels: *root*, *type*, and *specific*. Each of them corresponds to a different generalization level of assembly code. More details can be found in our technical report.\(^6\)

### 6. LOCALITY SENSITIVE HASHING

In this section, we introduce an Adaptive Locality Sensitive Hashing (ALSH) scheme for searching the block-level semantic clones. As discussed in Section 2, exact nearest neighbor search is not scalable. Thus, we started from the reduction of the \(\epsilon\)-approximated \(k\)-NN problem:

\[
(h(o_1) = h(o_2)) = 1 - \frac{\theta_{o_1, o_2}}{\pi}
\]

\(\theta_{o_1, o_2}\) is the included angle between \(o_1\) and \(o_2\). The probability that two vectors have the same projected direction on a random hyperplane is high when their included angle is small.

**Theorem 1.** The random hyperplane hash function is a \((r, cr, 1 - r/\pi, 1 - cr/\pi)\) sensitive hashing family. \(\square\)

**Proof.** According to Definition 4 and Equation 2: \(p_1 = 1 - r/\pi\) and \(p_2 = 1 - cr/\pi\). \(\square\)

To use the locality sensitive hashing families to solve the ball cover problem, it needs a hashing scheme to meet the quality requirement. The E2LSH approach was originally proposed by [14] and extended by [5]. It concatenates \(k\) different hash functions \([h_1, \ldots, h_k]\) from a given LSH family \(H\) into a function \(g(o) = (h_1(o), \ldots, h_k(o))\), and adopts \(l\) such functions. The parameters \(k\) and \(l\) are chosen to ensure the following two properties are satisfied:

**Property 1.** \((P_1)\): if there exists \(p^* \in B(q, r)\), then \(g_j(p^*) = g_j(q)\) for some \(j = 1 \ldots l\). \(\square\)

**Property 2.** \((P_2)\): the total number of points \(\notin B(q, cr)\) that collides with \(q\) is less than \(2l\). \(\square\)
It is proven that if the above two properties hold with constant probability, the algorithm can correctly solve the \((\epsilon, r)-\text{approximated ball cover problem}\) \cite{14}. For E2LSH, by picking \(k = \log_{\rho}(1/n)\) and \(l = n^\rho\) where \(\rho = \left\lceil \frac{\ln(1/p)}{\ln(1/\epsilon^2)} \right\rceil\), both Properties 1 and 2 hold with constant probability.

However, the ball cover problem is a strong reduction to the NN problem since it adopts the same radius \(r\) for all points. Real-life data cannot always be evenly distributed. Therefore, it is difficult to pick an appropriate \(r\). We denote this as the uneven data distribution issue. In \cite{12}, a magic \(r_m\) is adopted heuristically. But as pointed out by \cite{32}, such a magic radius may not exist. A weaker reduction was proposed in \cite{14}, where the NN problem is reduced to multiple \((r, \epsilon)\)-NN ball cover problems with varying \(r = \{1, \epsilon^2, \epsilon^3, \ldots\}\). The intuition is that points in different density areas can find a suitable \(r\). However, such a reduction requires a large space consumption and longer response time. Other indexing structures have been proposed to solve this issue. Per our discussion in Section 2, existing techniques do not meet our requirement. Thus, we customize the LSH-forest approach and propose the ALSH structure.

6.1 Adaptive LSH Structure

We found that the limitation of the expanding sequence of \(r\) in previous section is too strong. It is unnecessary to follow the sequence \(r = \{1, \epsilon^2, \epsilon^3, \ldots\}\) as long as \(r\) is increasing in a similar manner to \(r_{t+1} = r_t \times \epsilon\). Thus, we customize the \(\epsilon\)-approximated NN problem as follows:

\[
\text{Definition 5.} \quad \left( f(r) \text{-approximated NN search problem} \right) \quad \text{Given a dataset } D \subseteq \mathbb{R}^d \text{ and a query point } q, \text{ let } r \text{ denote the distance between the query point } q \text{ and its nearest neighbor } a^*.
\]

The problem is to find an approximated data point within the distance \(f(r)\), where \(f(r)/r > 1\)

Instead of using a fix approximation ratio, we approximate the search by using a function on \(r\). We issue a different sequence of expanding \(r\). The expanding sequence of \(r\) is formulated as \(r_0, r_1, \ldots, r_t, r_{t+1}, \ldots, r_m\), where \(r_t < r_{t+1}\). Similar to the E2LSH approach, we concatenate multiple hash functions from the random hyperplane hash family \(\mathcal{H}\) into one. However, we concatenate different number of hash functions for different values of \(r\). This number is denoted by \(k_t\) for \(r_t\) and the sequence of \(k\) is denoted by \(k_0, k_1, \ldots, k_t, k_{t+1}, \ldots, k_m\), where \(k_t > k_{t+1}\). Recall that the concatenated function is denoted by \(g\). Consequently, there will be a different function \(g\) at position \(t\), which is denoted by \(g_t\). Yet, function \(g_t\) and function \(g_{t+1}\) can share \(k_{t+1}\) hash functions. With \(p_m\) to be specified later, we set the \(r\) value at position \(t\) as follows:

\[
r_t = \pi \times \left(1 - \frac{p_m}{n}\right)
\]

This allows us to have the effect of increasing the \(r\) value by decreasing the \(k\) value. We calculate the value of \(k\) at position \(t\) as follows:

\[
k_t = c \times k_{t+1}, \text{ where } c > 1
\]

By getting \(t_{t+1}\) from Equation 3, substituting \(k_t\) using Equation 4, and substituting \(p_m\) using Equation 3, we have:

\[
r_{t+1} = \pi \times \left(1 - \frac{r_{t+1}}{r_t} \right) = f_c(r_t) \quad f_c(r_t)/r_t > 1
\]

By setting \(c\) equals to 2, we can get an approximately similar curve of \(r\) sequence to the original sequence \(r_{t+1} = r_t \times \epsilon\)

![Figure 4: The index structure for the Adaptive Locality Sensitive Hashing (ALSH). There are \(m + 1\) levels on this tree. Moving from level \(t\) to level \(t+1\) is equivalent of increasing the search radius from \(r_t\) to \(r_{t+1}\).](image)

where \(\epsilon\) equals to 2. Following the aforementioned logic, we construct an Adaptive Locality Sensitive Hashing (ALSH) index in the form of prefix trees.

As shown in Figure 4, the index structure is a prefix tree of the signature values calculated by \(G = \{g_m, g_{m-1}, \ldots, g_0\}\). Level \(t\) corresponds to the position \(t\) in the \(r\) expanding sequence. By introducing different values of \(k_t\), each level represents a different radius \(r_t\). Each level denotes a different \(g_t\) function and the \(g_t\) function is a concatenation of \(k_t\) hash functions. Moving up from a node at level \(t\) to its parent at level \(t+1\) indicates that it requires a shorter matched prefix. The nodes which have the same parent at level \(t\) share the same prefix that is generated by \(g_t\).

To locate the leaf for a given data point \(q \in \mathbb{R}^d\), ALSH dynamically constructs the hash functions by trying \(g_t \in G\) in sequence. The signature of \(g_t\) can be generated by padding additional hash values to \(g_{t+1}\) since \(k_t = c \times k_{t+1}\). Following \cite{14, 32}, with \(l\) to be specified later, we adopt \(l\) such prefix trees as the ALSH’s index. Given a query point \(q\), we first locate the corresponding leaves in all prefix trees. With \(l\) to be specified later, we collect the first \(2l\) points from all the leaf buckets. To index a point, we locate its corresponding leaf in each tree and insert it to the leaf bucket. Suppose a leaf is on level \(t\). If the number of points in that leaf is more than \(2l\), we split all the data points of that leaf into the next level \(t\) by using \(g_t\). All the trees are dynamically constructed based on the incoming points to be indexed in sequence. Therefore, they can be incrementally maintained. Unlike the learning-based LSH \cite{11}, Kamin0 does not require the whole repository to estimate the hash functions to build the index.

It can be easily proved that \(g_t\) is a \((r_t, r_{t+1}, p_m, p_m)\)-sensitive hash family and \(g_t\) can correctly solve the \((r_{t+1}/r_t, r_t)\)-approximated ball cover problem by setting \(p_m = 1/n\) and \(l = n^{1/c}\). The proof follows \cite{14}. Details and implementation on key-value data store can be found in our technical report. Another parameter \(r_m\) controls the starting radius value at the root value. It indicates the maximum distance that two points can be considered as valid neighbors. For sparse points far away from each other, they are not considered as neighbors unless their distance is within \(r_m\).

For a single ALSH tree, the depth in the worst case is \(k_0\), and all the leaves are at level 0. In this case, the tree is equivalent to the E2LSH with \(k = k_0\). Therefore, the
space consumption for \( l = n^{1/c} \) ALSH trees are bounded by \( O(n^c + n^{1+1/c}) \), where \( O(n^c) \) is the data points in a dataset and \( O(n^{1+1/c}) \) is the space of indexes for the trees. The query time for a single ALSH prefix tree is bounded by its height. Given the maximum \( k \) value \( k_0 \) and the minimum \( k \) value \( k_m \), its depth in the worst case is \( \log_x(k_0/k_m) + 1 \). Thus, the query time for \( l = n^{1/c} \) prefix trees is bounded by \( O((\log_x(k_0/k_m) \times n^{1/c})) \). The ALSH index needs to build \( n^{1/c} \) prefix trees for the full theoretical quality to be guaranteed. Based on our observation, setting \( l \) to 1 and 2 is already sufficient for providing good quality assembly code clones.

7. **SUBGRAPH CLONE SEARCH**

Even subgraph isomorphism is NP-hard in theory [21, 28], many algorithms have been proposed to solve it in a reasonable time. Formally, the subgraph isomorphism algorithm solved by most of these systems [13, 21, 31] is defined as:

**Definition 6. (Subgraph Isomorphism Search)** A graph is denoted by a triplet \((V, E, L)\) where \( V \) represents the set of vertices, \( E \) represents the set of edges, and \( L \) represents the labels for each vertex. Given a query graph \( q = (V', E', L') \) a subgraph isomorphism (also known as *embedding*) is an injective function \( M : V \to V' \) such that the following conditions hold: (1) \( \forall u \in V, L(u) \in L'(M(u)) \), (2) \( \forall \{u_i, u_j\} \in E, (M(u_i), M(u_j)) \in E' \), and (3) \( L(u_i, u_j) = L'(M(u_i), M(u_j)) \). The search problem is to find all distinct embeddings of \( q \) in \( g \).

The difference between this problem and ours in Definition 1 is two-fold. First, our problem is to retrieve *all the subgraph clones* of the target function \( f_s \)’s control flow graph from the repository. In contrast, this problem only needs to retrieve the exact matches of query graph \( q \) within \( g \). Refer to Conditions 1, 2, and 3 in Definition 6, or the termination condition of the procedure on Line 1 of subroutine SubgraphSearch in [21]. Our problem is more challenging and can be reduced to the problem in Definition 6 by issuing all the subgraphs of \( f_s \) as queries, which introduces a higher algorithmic complexity. Second, there is no such \( L \) data label attribute in our problem, but two types of edges: the control flow graph which links the basic blocks and the semantic relationship between basic blocks which is evaluated at the querying phase. Existing algorithms for subgraph iso-

**Algorithm 1 Mapper**

*Input* A basic block clone pair \((b_i, b_j)\)

*Output* A pair consisting of \((f_s, s_g)\)

1: \(f_s \leftarrow \text{getFunctionId}(b_i)\)
2: \(s_g \leftarrow []\) \(\triangleright\) create an empty list of subgraph clones
3: \(\text{cloneGraph} \leftarrow \{(b_i, b_j)\}\) \(\triangleright\) create a subgraph clone with one clone pair
4: \(s_g[0] \leftarrow \text{cloneGraph}\) \(\triangleright\) list of subgraph clones but at this moment, it has only one.
5: \(\text{return } (f_s, s_g)\) with \(f_s\) as key index

7.1 **MapReduce Subgraph Search**

We adopt two functions in the Apache Spark MapReduce execution framework, namely the *map* function and the *reduce-by-key* function. In our case, the map function transforms the clone pairs generated by ALSH (refer to the data flow in Figure 3) and the reduce-by-key function constructs subgraph clone respectively for different repository function \( f_s \) if it has more than one clone blocks in the previous step.

The outputs of the map functions correspond to the first row in Figure 5. A red circle represents a target basic block \( b_i \) and a green triangle represents a source basic block \( b_j \). The link between them indicates that they are a block-to-block clone pair \((b_i, b_j)\), which is produced in the previous step. A white rectangle represents a list of subgraph clones and the colored rectangle inside it represents a subgraph clone. Algorithm 1 maps each clone pair into a list of subgraph clones which contains only one subgraph clone. Each subgraph clone is initialized with only one clone pair.

After the map transformation functions, the reduce-by-key function reduces the produced lists of subgraph clones. The reducer merges a pair of lists \( s_g^1 \) and \( s_g^2 \) into a single one, by considering their subgraph clones’ connectivity. The reduce process is executed for the links from the second row to the last row in Figure 5. Only the lists of subgraph clones with the same \( f_s \) will be merged. As indicated by the links between the first and second row in Figure 5, only rectangles with the same orange background can be reduced together. Rectangles with other background colors are reduced with their own group.

Algorithm 2 shows the reduce function in details. Given two lists of subgraph clones under the same repository function \( f_s \), the reduce function compares their subgraph clones (Lines 1 and 2) and checks if two graphs can be connected (Lines 4 to 13) by referring to the the control flow graph edges \( E_c \) and \( E_r \). If the two subgraph clones can be connected by one clone pair, then they can be merged into a single one (Lines 6 and 7). If a subgraph clone from \( s_g^2 \) cannot be merged into any subgraph clones in \( s_g^1 \), it will be appended to the list \( s_g^1 \) (Lines 20 and 21). At the end of the
Figure 6: (a) the EDF function on repository function clone pair similarity, (b) the kernel density & histogram of the cosine similarity of each basic block’s 20th-nearest neighbor, (c) the EDF on per assembly function block count |B_s|, and (d) the kernel density & histogram on assembly function block count |B_s| < 40.

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Branch Count</th>
<th>Function Count</th>
<th>Block Count</th>
<th>Clone Pair Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>5</td>
<td>590</td>
<td>15,184</td>
<td>1,329</td>
</tr>
<tr>
<td>curl</td>
<td>16</td>
<td>9,268</td>
<td>176,174</td>
<td>40,317</td>
</tr>
<tr>
<td>expat</td>
<td>3</td>
<td>2,025</td>
<td>35,801</td>
<td>14,054</td>
</tr>
<tr>
<td>jsoncpp</td>
<td>14</td>
<td>11,779</td>
<td>43,734</td>
<td>204,701</td>
</tr>
<tr>
<td>ljpeg</td>
<td>9</td>
<td>4,605</td>
<td>82,257</td>
<td>18,946</td>
</tr>
<tr>
<td>libtiff</td>
<td>13</td>
<td>7,276</td>
<td>124,974</td>
<td>51,925</td>
</tr>
<tr>
<td>openssl</td>
<td>9</td>
<td>13,732</td>
<td>200,415</td>
<td>29,767</td>
</tr>
<tr>
<td>sqlite</td>
<td>12</td>
<td>9,437</td>
<td>202,777</td>
<td>23,674</td>
</tr>
<tr>
<td>tinyxml</td>
<td>7</td>
<td>3,286</td>
<td>30,401</td>
<td>22,798</td>
</tr>
<tr>
<td>zlib</td>
<td>8</td>
<td>1,741</td>
<td>30,585</td>
<td>6,854</td>
</tr>
<tr>
<td>total</td>
<td>96</td>
<td>69,939</td>
<td>942,299</td>
<td>423,365</td>
</tr>
</tbody>
</table>

Table 1: The assembly code clone dataset summary.

8. EXPERIMENTS

This section presents comprehensive experimental results for the task of assembly code clone search. First, we explore a graph search algorithm, we solve the problem in Definition 1. In order to obtain a ranked list of repository functions for f_s, we calculate the similarity value by checking how much its subgraphs s_g cover the graph of the query f_s: sim_m = ⟨|uniqueEdges(s_g) + uniqueNodes(s_g)| / |B_s| + |E_l⟩.

Compared to other join-based or graph-exploration-based search approach, our MapReduce-based search procedure avoids recursive search and is bounded by polynomial-complexity. Let m_s be the number of clone pairs for a target function f_s. There are at most O(m_s^2) connectivity checks between the clone pairs (no merge can be found) and the map function requires O(m_s) executions. m_s corresponds to the number of rectangles in the second row of Figure 5. Refer from the second row to the last one. The reduce function is bounded by O(m_s^2) comparisons. m_s is bounded by O(|B_s| × |B_s|), which implies that each basic block of f_s is a clone with all the basic blocks of f_s. However, this extreme case rarely happens. Given the nature of assembly functions and search scenarios, m_s is sufficiently bounded by O(max(|B_s|, |B_s|)). According to the descriptive statistics of our experiment, 99% of them have less than 200 basic blocks.

8.1 Labeled Dataset Generation

One of the challenging problems for assembly code clone search is the lack of labeled (ground truth) dataset, since the most effective labeled dataset requires intensive manual identification of assembly code clones [7]. To facilitate future studies on assembly clone search, we have developed a tool to systematically generate the assembly function clones based on source code clones. The tool performs four steps: Step 1: Parse all the source code functions from different branches or versions of a project and identify all the function-to-function clones using CCFINDERX [16], which estimates source code similarity based on the sequence of normalized tokens. Step 2: Compile the given branches or versions of a project with an additional debug flag to enable the compiler output debug symbols. Step 3: Link the source code functions to the assembly code functions using the compiler output debug symbols, where such information is available. Step 4: For each pair of source code clone, generate a pair of assembly function clone and transfer the similarity to the new pair.

The intuition is that the source code function-level clones indicate the functional clones between their corresponding assembly code. In [7, 25], the source code and assembly code are manually linked with an injected identifier in the form of variable declarations. However, after the automation of such process, we find that the link rate is very low due to the impact of compiler optimizations. The generated...
assembly code clone is in fact the combined result of source code patches and compiler optimizations. The source code evolves from version to version, and different versions may have different default compiler settings. Thus, the labeled code patches and compiler optimizations. The source code assembly code clone is in fact the combined result of source

Table 2: Benchmark results of different assembly code clone search approaches. We employed three evaluation metrics: the Area Under the Receiver Operating Characteristic Curve (AUROC), the Area Under the Precision-Recall Curve (AUPR), and the Mean Average Precision at Position 10 (MAP@10). O denotes that the method is not scalable and we cannot obtain a result for this dataset within 24 hours.

Table 3: Paired t-test on the normalization level.

Assembly code normalization is used in [7, 27]. However, its effects were not formally studied. In this section, we present the results of the statistical test on the effects of the normalization level. Details on normalization can be found in our technical report. We start by using a strong hash code clone search with different normalization levels on each of the generated datasets. Then, we collect the corresponding precision value to the given normalization level as samples and test the relationship between precision and the chosen normalization level. Normalization can increase the recall, but we want to evaluate the trade-off between the precision and different normalization levels. According to the ANOVA
we employ all of them. AUROC and AUPR can test the classifier by issuing different threshold values consecutively [9, 24, 30], while MAP@10 can evaluate the quality of the top-ranked list simulating the real user experience [23]. Table 2 presents the benchmark results. The highest score of each evaluation metric is highlighted for each dataset. Also, the micro-average of all the results for each approach is given in the Avg column. Kam1n0 out-performs the other approaches in almost every case for all evaluation metrics. Kam1n0 also achieves the best averaged AUROC, AUPRC, and MAP@10 scores. The overall performance also suggests that it is the most stable one. Each approach is given a 24-hour time frame to finish the clone search and it is only allowed to use a single thread. Some results for BinClone and Tracelet are empty, which indicate that they are not scalable enough to obtain the search result within the given time frame. Also, we notice that BinClone consumes more memory than the others for building the index, due to its combination of features which enlarges the feature space. We notice that the experimental results are limited within the context of CCFinder. In the future, we will investigate other source code clone detection techniques to generate the ground truth data.

8.4 Scalability Study

In this section, we evaluate Kam1n0’s scalability on a large repository of assembly functions. We set up a mini-cluster on Google Cloud with four computational nodes. Each of them is a n1-highmem-4 machine with 2 virtual cores and 13 GB of RAM. We only use regular persistent disks rather than solid state drives. Each machine is given 500 GB of disk storage. All the machines run on CentOS. Three machines run the Spark Computation Framework and the Apache Cassandra Database and the other runs our Kam1n0 engine. To conduct the experiment, we prepare a large collection of binary files. All these files are either open source libraries or applications, such as Chromium. In total, there are more than 2,310,000 assembly functions and 27,666,692 basic blocks. Altogether, there are more than 8 GB of assembly code. We gradually index this collection of binaries in random order, and query the zlib binary file of version 2.7.0 on Kam1n0 at every 10,000 assembly function indexing interval. As zlib is a widely used library, it is expected that it has a large number of clones in the repository. We collect the average indexing time for each function to be indexed, as well as the average time it takes to respond to a function query. Figure 7 depicts the average indexing and query response time for each function. The two diagrams suggest that Kam1n0 has a good scalability with respect to the repository size. Even as the number of functions in the repository increases from 10,000 to 2,310,000, the impact on the response time is negligible. There is a spike up at 910,000 due to the regular compaction routine in Cassandra, which increases I/O contention in the database.

9. CONCLUSION AND LESSON LEARNED

Through the collaboration with Defence Research and Development Canada (DRDC), we learned that scalability, which was not considered in previous studies, is a critical issue for deploying a successful assembly clone search engine. To address this, we present the first assembly search engine that combines LSH and subgraph search. Existing off-the-shelf LSH nearest neighbor algorithms and subgraph isomorphism search techniques do not fit our problem setting. Therefore, we propose new variants of LSH scheme and incorporate it
with graph search to address the challenges. Experimental results suggest that our proposed MapReduce-based system, Kam1n0, is accurate, efficient, and scalable. Currently Kam1n0 can only identify clones for x86/amd64 processor. In the future, we will extend it to the other processors and investigate approaches that can find clones between different processors. Kam1n0 provides a practical solution of assembly clone search for both DRDC and the reverse engineering community. The contribution is partially reflected by the award received at the 2015 Hex-Rays Plug-In Contest.

10. ACKNOWLEDGMENT

The authors would like to thank the reviewers for the thorough reviews and valuable comments. This research is supported by Defence Research and Development Canada (contract no. W7701-155902/001/QCL) and Canada Research Chairs Program (950-230623).

11. REFERENCES


